PENYELESAIAN *BOUNDED KNAPSACK PROBLEM* MENGGUNAKAN *DYNAMIC PROGRAMMING* (Studi Kasus: CV. Mulia Abadi)

Sandi Kosasi

Program Studi Teknik Informatika
Sekolah Tinggi Manajemen Informatika dan Komputer Pontianak
Jln. Merdeka No. 372 Pontianak, Kalimantan Barat
Email: sandykosasi@stmikpontianak.ac.id

ABSTRACT

Knapsack problem merupakan masalah optimasi kombinasi dengan tujuan memaksimalkan total nilai dari barang-barang yang dimasukkan ke dalam *knapsack* atau suatu wadah tanpa melewati kapasitasnya. Penelitian menekankan kepada *bounded knapsack problem* yang merupakan pengembangan dari *0-1 knapsack problem* menggunakan algoritma *dynamic programming*. Penelitian berbentuk studi kasus dengan metode *quasi eksperimental*. Pengumpulan data menggunakan wawancara dan observasi. Untuk analisis dan perancangannya menggunakan metode OOAD (*Object-Oriented Analysis and Design*) dan pengujiannya menggunakan model V. Aplikasi ini dikembangkan dengan bahasa pemrograman Java dengan kemampuan menentukan nilai prioritas tertinggi berdasarkan daftar barang dan harga yang optimal sesuai dengan anggaran belanja. Aplikasi ini mudah digunakan oleh pembeli, mulai dari memasukan kombinasi dari sejumlah daftar barang belanjaan yang dibutuhkan dengan batasan dari jumlah anggaran yang tersedia.

Kata Kunci: *knapsack problem, 0-1 knapsack probelm, dynamic programming, object-oriented analysis and design, V Model*

PENDAHULUAN

CV. Mulia Abadi merupakan sebuah perusahaan yang menjual berbagai macam makanan dan barang kelontong untuk kawasan masyarakat di kota Pontianak dan sekitarnya. Dari setiap kali transaksi jual beli, kecenderungan yang terjadi adalah para pembeli mengalami kesulitan menentukan kombinasi yang optimal mengenai banyaknya barang yang akan dibeli namun dengan anggaran belanja yang terbatas. Selama ini proses penentuan kombinasi optimal tersebut dilakukan tanpa menggunakan komputasi dan dengan terbatasnya waktu dan tenaga sehingga hanya menghasilkan keputusan yang mendekati nilai optimal. Kenyataan ini jelas dapat merugikan kedua belah pihak, dari sisi pembeli dan penjual. Persoalan dalam menentukan kombinasi barang yang akan dipilih yang dapat memberikan hasil yang maksimal dengan tidak melebihi anggaran belanja adalah merupakan inti dari *knapsack problem*. Dimana tiap barang memiliki nilai harga, nilai prioritas, dan jumlah satuan untuk tiap barang yang tersedia.

Penyelesaian *knapsack problem* menggunakan *dynamic programming* dimana memiliki tahapan keputusan yang saling berhubungan untuk mencari solusi dengan kombinasi yang optimal [4]. *Dynamic programming* merupakan metode pemecahan masalah dengan cara menguraikan solusi menjadi sekumpulan langkah (*step*) atau tahapan (*stage*) sedemikian sehingga solusi dari persoalan dapat dipandang dari serangkaian keputusan yang saling berkaitan [6]. *Dynamic programming* akan bekerja dengan memecahkan suatu masalah menjadi beberapa tahapan atau sub masalah dan menyelesaikannya satu persatu, mulai dari yang paling sederhana dan menggunakan jawaban dari tahapan tersebut untuk menyelesaikan tahapan berikutnya yang lebih rumit. Jenis dari *knapsack problem* yang digunakan adalah *Bounded Knapsack Problem*.

Bounded knapsack problem ini merupakan pengembangan dari *0-1 knapsack problem*, dimana jumlah barang untuk tiap jenis item barang yang akan dipilih terbatas jumlahnya, bukan merupakan bilangan negatif, dan tiap barang yang akan dipilih haruslah tetap dalam keadaan utuh, bukan merupakan fraksi atau sebagian dari kesatuan barang tersebut [6]. Aplikasi ini dapat membantu pembeli, dimana mereka dapat lebih mudah dan efektif dalam menemukan solusi untuk menentukan setiap jenis dan item barang yang dapat dibeli. Melalui kombinasi yang optimal dari setiap jenis dan item barang dengan jumlah satuannya yang memiliki jumlah nilai prioritas tertinggi, dan dengan jumlah harga barang tidak melewati jumlah anggaran belanja yang tersedia. Semakin besar data yang digunakan, maka waktu yang dibutuhkan algoritma untuk menyelesaikan

juga semakin lama. Pengujiannya menggunakan cek solusi *knapsack problem*, lama waktu proses, analisis kompleksitas ruang dan waktu dan analisis efisiensi dengan algoritma rekursif.

TINJAUAN PUSTAKA

Knapsack Problem

Knapsack problem merupakan masalah optimasi kombinasi dengan tujuan untuk memaksimalkan total nilai dari barang-barang yang dimasukkan ke dalam *knapsack* atau suatu wadah tanpa melewati kapasitas wadah tersebut. *Knapsack problem* atau *rucksack problem* adalah masalah optimasi kombinatorial. Namanya berasal dari masalah maksimasi untuk pilihan paling tepat dari barang-barang yang akan dibawa dalam sebuah tas pada sebuah perjalanan [4]. Sejumlah barang yang tersedia ini, masing-masing memiliki berat dan nilai, yang menentukan jumlah barang yang dapat dibawa sehingga total berat tidak melebihi kapasitas tas dan dengan total nilai yang sebesar mungkin. *Knapsack problem* dapat didefinisikan sebagai sebagai sebuah perumpamaan dari *knapsack problem* dengan set item N, terdiri dari n item j dengan keuntungan pj dan berat wj, dan nilai kapasitas c. Kemudian tujuannya adalah untuk memilih sebuah bagian dari N dimana total keuntungan dari item yang dipilih dimaksimalkan dan total berat tidak melebihi c. Selanjutnya *0-1 knapsack problem* adalah masalah dalam memilih sebuah bagian dari n buah barang i yang memiliki jumlah keuntungan maksimal dengan jumlah berat barang tidak melebihi kapasitas c. Permasalahan ini dapat dirumuskan sebagai berikut [4]:

Maksimalkan

$$\sum_{j=1}^{n} p_j \, x_j \qquad (2.1)$$

Batasan

$$\sum_{j=1}^{n} w_j \, x_j \leq c \qquad (2.2)$$

xj $\in$ {0,1} j = 1, . . . , n

pj, wj, xj, dan c merupakan bilangan bulat positif

Keterangan :

n = jumlah barang
j = nama barang
pj = profit atau keuntungan barang j
xj = jumlah barang j yang dimasukkan ke dalam knapsack
wj = berat barang j
c = kapasitas maksimal knapsack

Penggunaan *bounded knapsack problem* dimana jumlah barang untuk tiap barang yang tersedia terbatas jumlahnya. Barang-barang yang dimasukkan adalah barang yang berbentuk satuan dan tidak bisa dipecah menjadi beberapa bagian sehingga jika ingin memasukkan barang tersebut, maka satu kesatuan barang harus masuk ke dalam wadah. Dalam masalah ini, anggaran belanja yang dibawa oleh sang pembeli sejumlah U, dimana terdapat n buah barang i berbeda yang dapat dibeli dengan anggaran tersebut. Barang i memiliki harga hi dan nilai prioritas pi. Jika xi adalah jumlah barang i yang akan dibeli dengan anggaran yang tersedia. Tiap i memiliki batas atas bi dan batas bawah 0 (nol). Barang i dapat berulang kali dimasukkan dalam daftar beli hanya jika kedua kondisi berikut dipenuhi, yaitu: Jumlah barang tersebut tidak melebihi jumlah barang yang tersedia dan Barang yang dimasukkan dalam daftar beli tidak melanggar batasan kapasitas. Secara formal, *bounded knapsack problem* terdiri dari sekelompok barang {1,...,i} dengan jumlah harga barang yang tidak boleh melebihi jumlah uang U, dapat dirumuskan menjadi berikut [4]:

Maksimalkan

$$\sum_{i=1}^{n} p_i \, x_i \qquad (2.3)$$

Batasan

$$\sum_{i=1}^{n} h_i \, x_i \leq U \qquad (2.4)$$

dengan $b_i \geq x_i \geq 0$

i = 1, . . . , n

p_i, h_i, b_i, U, dan x_i adalah bilangan bulat positif

Keterangan :

U = jumlah uang
i = nama barang
n = jumlah barang
h_i = harga barang i
p_i = nilai prioritas barang i
x_i = jumlah barang i yang akan dibeli
b_i = nilai batas atas barang i

Dynamic Programming

Dynamic programming ialah suatu metode membuat tahapan keputusan yang saling berhubungan untuk mencari solusi dengan kombinasi yang optimal [2]. Kata *programming* pada istilah *dynamic programming* tidak memiliki hubungan dengan menulis kode apa pun atau program komputer, melainkan sebuah sinonim untuk optimasi dan memiliki arti sebagai perencanaan atau sebuah metode tabulasi. *Dynamic programming* adalah strategi untuk membangun masalah optimasi bertingkat, yaitu masalah yang dapat digambarkan dalam bentuk serangkaian tahapan (*stage*) yang saling mempengaruhi [6]. *Dynamic programming* merupakan metode untuk menyelesaikan secara efisien masalah pencarian dalam jangkauan luas dan optimasi yang memiliki karakteristik *overlapping subproblem* dan *optimal substructure* [2].

Penyelesaian masalah dalam *dynamic prorgramming* dilakukan secara rekursif yang berarti setiap kali mengambil keputusan harus memperhatikan keadaan yang dihasilkan oleh keputusan optimal dari tahap sebelumnya dan merupakan landasan bagi keputusan optimal pada tahap berikutnya [2]. Prosedur mencari nilai dari penyelesaian optimal dalam bentuk penyelesaian optimal dari submasalah secara rekursif. Pada rekursif akan dilakukan pemanggilan prosedur atau fungsi yang sama. Dalam tiap rekursif, sebuah barang akan dimasukkan atau dikeluarkan dari daftar barang yang akan dibeli. *Bounded knapsack problem* akan diubah ke bentuk *0-1 knapsack problem* yang setara berdasarkan nilai batas atas untuk setiap barang i. Rumus rekursif untuk *0-1 knapsack problem* [5]:

$$z[k,h] = \begin{cases} 0 & if\ k = 0\ atau\ h = 0 \\ z[k-1,h] & if\ h < h_k \\ \max\{z[k-1,h],\ z[k-1,h-h_k] + p_k\} & if\ h \geq h_k \end{cases} \quad (2.5)$$

Keterangan :

z = nilai optimal dari fungsi tujuan
h = jumlah harga barang maksimal yang diperbolehkan
k = barang ke-k
hk = harga barang k
pk = nilai prioritas barang k

Rumus 2.5 dapat dijelaskan sebagai berikut, z(k,h) menunjukan nilai prioritas tertinggi yang didapatkan jika anggaran yang tersedia adalah U dan terdapat sejumlah barang dari 1,...., n. Tujuan dari perhitungan tersebut ialah menemukan sebuah bagian dari n barang yang memiliki jumlah prioritas maksimal atau tertinggi tanpa melebihi nilai U. Pada tiap tahap (misal: k) akan dilakukan proses memasukkan barang ke dalam solusi daftar beli yang disesuaikan dengan status (misal: h) jumlah harga barang yang diperbolehkan atau tersisa setelah memasukkan barang pada tahap sebelumnya, sampai batas anggaran uang maksimumnya. Aturan pertama k = 0, maka hasil perhitungan akan menghasilkan nilai 0, karena berarti tidak ada barang yang dapat dimasukkan ke dalam solusi daftar beli. Pada h = 0, hasil perhitungan juga akan menghasilkan nilai 0, karena berarti jumlah harga barang yang diperbolehkan adalah 0. Pada aturan kedua, Jika harga barang k melebihi jumlah harga barang yang diperbolehkan, maka barang k tidak akan dimasukkan ke dalam solusi daftar beli. Sedangkan pada aturan ketiga jika harga barang k kurang dari atau sama dengan jumlah harga barang yang diperbolehkan, maka barang k memiliki dua kemungkinan, dimasukkan atau tidak dimasukkan ke dalam solusi daftar beli. Ketika memasukkan suatu barang pada tahap k, maka jumlah harga barang yang dapat dibeli sekarang adalah h-hk [5].

Prinsip optimalitas akan diterapkan dalam memakai jumlah harga barang yang tersisa, dengan mengacu pada nilai optimum prioritas dari tahap sebelumnya. Jika dengan memasukkan barang tersebut, jumlah nilai prioritas lebih besar daripada jumlah nilai prioritas tanpa memasukkan barang k, maka keputusan optimalnya adalah memasukkan barang tersebut ke dalam solusi daftar beli. Jika yang terjadi adalah kebalikannya, maka barang k tidak akan dimasukkan ke dalam solusi daftar beli. Dari beberapa kemungkinan solusi yang ada, hanya akan dipilih solusi yang memiliki prioritas yang lebih tinggi. Jika jumlah nilai prioritas pada suatu tahap sama dengan jumlah nilai prioritas tahap sebelumnya, maka akan diprioritaskan untuk menggunakan keputusan optimal pada tahap sebelumnya. Algoritma *Dynamic programming* yang dibangun menggunakan solusi rekursif berikut [6]:

```
def A(w, v, i,j):
   if i == 0 or j == 0: return 0
   if w[i-1] > j:  return A(w, v,
       i-1, j)
   if   w[i-1]   <=   j:   return
max(A(w,v,  i-1,  j),  v[i-1]  +
A(w,v, i-1, j - w[i-1]))
```

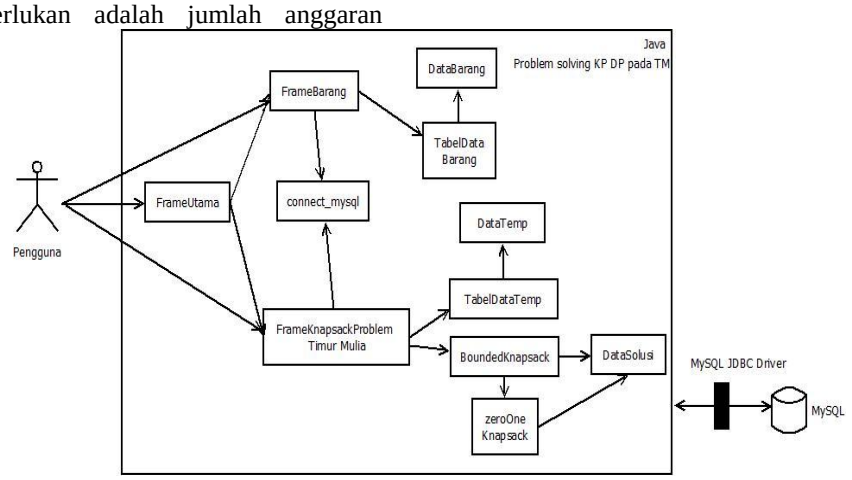
METODE PENELITIAN

Penelitian ini berbentuk studi kasus dengan metode *quasi eksperimental*. Metode pengumpulan data menggunakan wawancara dan observasi. Wawancara dengan sejumlah responden terpilih dengan menggunakan teknik *purposive sampling* [3]. Respondenya berasal dari pimpinan, manajemen dan staf/karyawan. Metode analisis dan perancangan menggunakan metode OOAD (*Object-Oriented Analysis and Design*). Untuk tahap analisis, metode ini memeriksa *requirement* yang harus dipenuhi sebuah sistem dari sudut pandang kelas-kelas dan objek-objek yang ditemui dalam ruang lingkup perusahaan. Sementara tahap desain, metode ini untuk mengarahkan arsitektur software yang didasarkan pada manipulasi objek-objek sistem atau subsistem [1]. Menggunakan pengujian model V dalam menjelaskan hubungan antara setiap langkah dengan tingkat pemenuhan kebutuhan [7]. Semakin menuju ke akhir proses, kebutuhan semakin dipenuhi. Kebutuhan diwakili dengan proses pengujian yang dilakukan setelah pengkodean dilakukan.

HASIL DAN PEMBAHASAN

Hasil penelitian ini memperlihatkan sebuah aplikasi yang dibangun dengan menerapkan *dynamic programming* untuk menyelesaikan *knapsack problem* pada CV. Mulia Abadi. Model penyelesaian terdiri dari tiga *frame* yaitu:

a. *Frame* utama merupakan form yang pertama kali dimunculkan saat menjalankan *problem solving* ini. Pada form utama berisi pilihan menu untuk membuka form barang dan form *knapsack problem*.
b. *Frame* barang merupakan form dimana pengguna dapat menambahkan data barang baru, mengubah dan menghapus data barang yang telah ada.
c. *Frame Knapsack Problem* merupakan form dimana pengguna dapat memasukkan data untuk masalah *knapsack* baru. Data masukan yang diperlukan adalah jumlah anggaran belanja yang tersedia dan daftar belanja sementara yang ingin dibeli oleh pembeli. Data daftar belanja sementara yang telah dimasukkan akan ditampilkan oleh tabel data *temp*. Setelah proses input data *temp* selesai, pengguna dapat mencari solusi penyelesaian masalah. Karena jenis masalah *knapsack problem* pada adalah *bounded knapsack problem*, maka data *temp* tersebut harus diubah terlebih dahulu menjadi *0-1 knapsack problem* oleh *Bounded Knapsack* kemudian diselesaikan menggunakan *dynamic programming* pada *ZeroOneKnapsack* dengan menggunakan *dynamic programming*. Penyelesaian *dynamic programming* ini akan ditampilkan pada daftar barang yang disarankan (Gambar 1).



Gambar 1. Arsitektur Penyelesaian *Knapsack Problem*

Fungsi utama dari *problem solving* ini adalah sebagai alat untuk menyelesaikan *knapsack problem*. *Problem solving* ini dirancang agar pengguna dapat lebih mudah memasukkan barang-barang yang diinginkan pembeli karena data barang yang dapat dibeli telah dimasukkan ke dalam basis data yang terhubung ke *problem solving* dan tersedia fungsi untuk mengubah maupun menghapus barang dari daftar barang. *Problem solving* ini juga menggunakan *dynamic programming* untuk membantu menyelesaikan masalah *bounded knapsack*. Metode *dynamic programming* yang digunakan memiliki beberapa tahap, antara lain:

a. Menentukan struktur dari masalah
   Suatu masalah akan dibagi menjadi beberapa tahap (*stage*). Tiap tahap terdiri dari sejumlah status (*state*) yang berhubungan dengan tahap tersebut. Secara umum, status merupakan bermacam kemungkinan solusi pada tahap tersebut. Misalnya terdapat suatu *knapsack problem* dimana jumlah uang maksimal (U) adalah 800 dan terdapat 5 jenis barang (n), dimana batas atas (bi) atau jumlah untuk tiap jenis barang adalah 1. Tiap barang memiliki nilai prioritas (pi) 2,1,2,3,6 dan harga (hi) 3,2,1,2,8 (dalam ratusan). Inti *knapsack problem* tersebut ialah menentukan optimasi kombinasi barang dan jumlah satuannya yang memiliki jumlah nilai prioritas tertinggi untuk dibeli dengan jumlah harga barang tidak melewati jumlah uang yang dimiliki pembeli. Total prioritas optimal (z) akan dicari menggunakan *dynamic programming*. Banyaknya tahap pencarian berdasarkan jumlah harga maksimal dan jumlah satuan barang. Jumlah harga maksimal (h) pada *knapsack problem* adalah jumlah uang maksimal dibagi 100, karena satuan harga terkecil barang dalam ratusan. Jumlah satuan barang (k) adalah jumlah dari barang (i) dikali batas atas barang (bi). Jadi, nilai h adalah 8 dan jumlah satuan barang (k) adalah 5. Perhitungan dimulai dengan mencari nilai prioritas satuan barang ke-1 lalu dilanjutkan mencari nilai prioritas satuan barang ke-2 dan

seterusnya sampai satuan barang ke-k. Hasil perhitungan total prioritas pada tiap tahap akan dimasukkan ke tabel pengingat dengan ukuran (k,h) (tabel1).

Tabel 1. Pengingat Dynamic Programming Sebelum Pencarian Solusi

	0	1	2	3	4	5	6	7	8
TIDAK ADA									
(P : 2, H: 3)									
(P : 1, H: 2)									
(P: 2, H: 1)									
(P: 3, H: 2)									
(P: 6,H: 8)									

b. Menentukan persamaan rekursif
Persamaan rekursif yang digunakan adalah persamaan rekursif 2.5 untuk *0/1 knapsack problem*.

c. Menghitung nilai dari solusi optimal
Pada setiap tahap akan dilakukan perhitungan sesuai fungsi rekursif untuk mencari solusi optimal yang memiliki nilai prioritas tertinggi dari satu atau lebih kemungkinan solusi yang ada. Pencarian solusi optimal tersebut dilakukan dengan memperhatikan keadaan pada tiap tahap dan menggabungkan solusi-solusi optimal dari tahap-tahap sebelumnya yang lebih kecil untuk mendapatkan solusi-solusi optimal untuk tahap-tahap berikutnya. Jika solusi optimal untuk suatu tahap telah ditemukan maka dilakukan perhitungan untuk tahap berikutnya. Hal ini akan terus dilakukan hingga ditemukan solusi optimal secara keseluruhan untuk masalah sebenarnya. Jika tidak ada barang apapun yang dimasukkan ke dalam daftar solusi atau k = 0, maka nilai optimal pada tahap z(0,h) = 0 (tabel 2).

Tabel 2. Pengingat Dynamic Programming Pencarian Solusi pada Baris Pertama

	0	1	2	3	4	5	6	7	8
TIDAK ADA	0	0	0	0	0	0	0	0	0
(P : 2, H: 3)									
(P : 1, H: 2)									
(P: 2, H: 1)									
(P: 3, H: 2)									
(P: 6,H: 8)									

Jika jumlah harga barang maksimal atau h = 0, maka nilai optimal pada tahap z (k,0) = 0. Contohnya pada tahap z (1,0) dapat dilihat pada gambar berikut (tabel 3).

Tabel 3. Pengingat Dynamic Programming Tahap z(1,0)

	0	1	2
TIDAK ADA	0	0	0
(P : 2, H: 3)	0		
(P : 1, H: 2)			

Jika harga barang melebihi jumlah harga maksimal, maka barang tersebut tidak dapat dibeli dan nilai optimal pada tahap tersebut bernilai z (k-1,h) yang berarti mengambil nilai optimal dari tahap sebelumnya yang tidak memasukkan barang tersebut. Contohnya pada tahap z (1,1), dimana harga barang ke-1 adalah 3 dan jumlah harga maksimal adalah 1. Nilai optimal pada tahap z (1, 1) adalah nilai optimal dari tahap sebelumnya yaitu nilai optimal tahap z (0,1) (tabel 4).

Tabel 4. Pengingat Dynamic Programming Tahap z(1,1)

	0	1	2
TIDAK ADA	0	0	0
(P : 2, H: 3)	0	0	
(P : 1, H: 2)			

Jika jumlah harga barang maksimal lebih dari atau sama dengan harga barang k, maka akan dilakukan perhitungan untuk mencari solusi optimal
z (k,h) = max {z (k-1, h), z (k-1, h – hk) + pk}. Contohnya pada tahap z (1,3), perhitungannya sebagai berikut (tabel 5):

z (k,h) = max {z (k-1, h), z (k-1, h – hk) + pk}
z (1,3) = max { z (0, 3), z (0, 3-3) + 2}
= max { z(0, 3), z (0, 0) + 2}
= max {0, 0 + 2}
= max {0, 2}
= 2

Tabel 5. Pengingat Dynamic Programming Tahap z(1,3)

	0	1	2	3	4	
TIDAK ADA	0	0	0	0	0	
(P : 2, H: 3)	0	0	0	2		
(P : 1, H: 2)						

Contoh lainnya pada tahap z(3,4), dynamic programming akan membandingkan nilai optimal antara tidak memasukkan barang tersebut (nilai optimal pada tahap z(2,4)) dengan memasukkan barang tersebut (nilai optimal pada tahap z(2,3) + 2) dan mengambil keputusan berdasarkan nilai optimal tertinggi. Jadi, keputusan terbaik pada tahap z(3,4) adalah memasukkan barang tersebut.

Perhitungannya sebagai berikut (tabel 6):
z (3,4) = max {z (k-1, h), z (k-1, h – hk) + pk}
= max {z (3-1, 4), z (3-1, 4-1) + 2}
= max {z (2, 4), z (2, 3) + 2}
= max {2, 4}
= 4

Tabel 6. Pengingat Dynamic Programming Tahap z(3,4)

	0	1	2	3	4	5
TIDAK ADA	0	0	0	0	0	0
(P : 2, H: 3)	0	0	0	2	2	2
(P : 1, H: 2)	0	0	1	2	2	3
(P: 2, H: 1)	0	2	2	3	4	
(P: 3, H: 2)						

Pencarian nilai solusi optimal akan terus dilakukan hingga tahap z(5,8). Berikut ilustrasi tabel pengingat dynamic programming tahap z(5,8).

Tabel 7. Pengingat Dynamic Programming Tahap z(5,8)

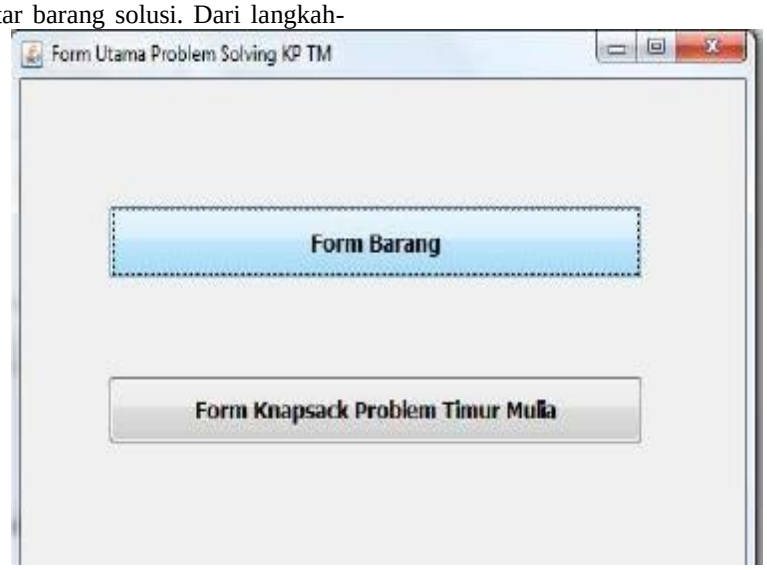
	0	1	2	3	4	5	6	7	8
TIDAK ADA	0	0	0	0	0	0	0	0	0
(P : 2, H: 3)	0	0	0	2	2	2	2	2	2
(P : 1, H: 2)	0	0	1	2	2	3	3	3	3
(P: 2, H: 1)	0	2	2	3	4	4	5	5	5
(P: 3, H: 2)	0	2	3	5	5	6	7	7	8
(P: 6,H: 8)	0	2	3	5	5	6	7	7	8

d. Menentukan keputusan optimal
Ilustrasi tabel pengingat berisi kemungkinan-kemungkinan yang diperoleh untuk tiap jumlah harga maksimal. Pada ilustrasi tabel 7 tersebut, total prioritas optimal yang didapat adalah z(5,8) = 8. Nilai prioritas tertinggi tersebut diambil dari nilai prioritas pada tahap terakhir. Penentuan tahap terakhir berdasarkan pada barang satuan terakhir dengan jumlah harga maksimal. Selanjutnya dilakukan perhitungan untuk menentukan barang-barang yang masuk solusi optimal. Untuk barang ke-k hingga k = 0, jika z(k,h) tidak sama dengan z(k-1,h) maka masukkan barang ke dftBrg(k) = 1.

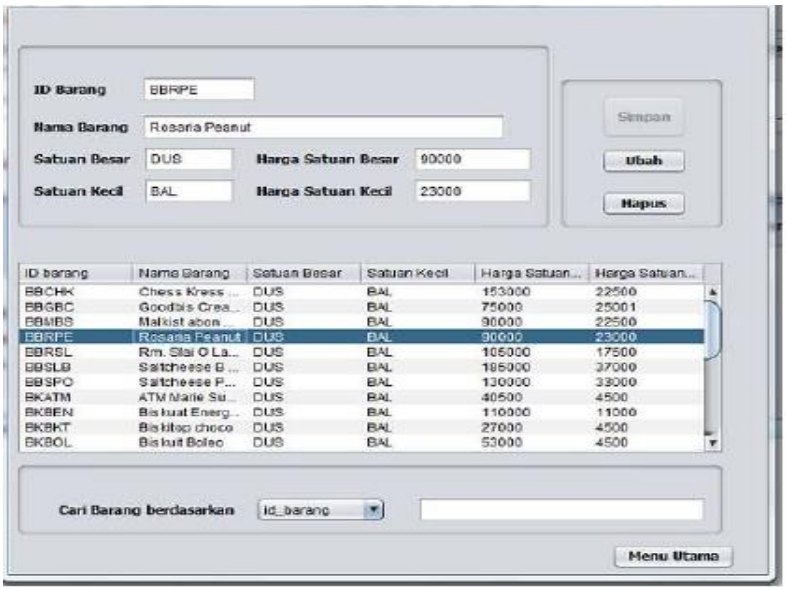
Menggunakan tabel 7, penentuan barang-barang yang masuk solusi dimulai dari barang satuan terakhir yaitu barang ke-5 dengan jumlah harga maksimal 8. Nilai prioritas optimal sebesar 8 pada z(5,8) dibandingkan dengan nilai 8 pada z(4,8). Karena nilainya sama, maka barang ke-5 tidak dimasukkan ke daftar barang solusi dan jumlah harga maksimal tidak berubah. Kemudian

nilai prioritas optimal barang ke-4 sebesar 8 pada z(4,8) dibandingkan dengan nilai 5 pada z(3,8). Karena nilainya berbeda, maka barang ke-4 dimasukkan ke daftar barang solusi. Setelah itu jumlah harga maksimal 8 dikurangkan dengan harga barang ke-4. Untuk perhitungan barang ke-3 bandingkan nilai 5 pada z(3,6) dengan nilai yang berada pada sebaris diatasnya yaitu 3 pada z(2,6). Karena nilainya berbeda, maka barang ke-3 dimasukkan ke daftar barang solusi. Setelah itu jumlah harga maksimal 6 dikurangkan dengan harga barang ke-3. Untuk perhitungan barang ke-2 bandingkan nilai 3 pada z(2,5) dengan nilai 2 pada z(1,5). Karena nilainya berbeda, maka barang ke-2 dimasukkan dan jumlah harga maksimal 5 dikurangkan dengan harga barang ke-2. Selanjutnya bandingkan nilai 2 pada z(1,3) dengan nilai 0 pada z(0,3). Karena nilainya berbeda, maka barang ke –1 dimasukkan ke daftar barang solusi. Dari langkah-langkah perhitungan tersebut diperoleh daftar barang solusi yang terdiri dari 1 barang ke-4, 1 barang ke-3, 1 barang ke-2 dan 1 barang ke-1.

Aplikasi ini dikembangkan dengan bahasa Java dan memiliki kemampuan dalam menentukan nilai prioritas tertinggi berdasarkan daftar barang dan harga yang optimal sesuai dengan anggaran belanja. Aplikasi ini memiliki sejumlah form antarmuka yang mudah untuk dapat digunakan oleh pembeli, mulai dari memasukan kombinasi dari sejumlah daftar barang belanjaan yang dibutuhkan dengan batasan dari jumlah anggaran yang tersedia. Aplikasi ini dibangun dengan 2 form utama yaitu form barang dan form knapsack problem CV. Mulia Abadi (gambar 2). Sebagian dari proses aplikasi *bounded knapsack problem* ini akan ditampilkan dalam beberapa gambar berikut ini (gambar 3-13).

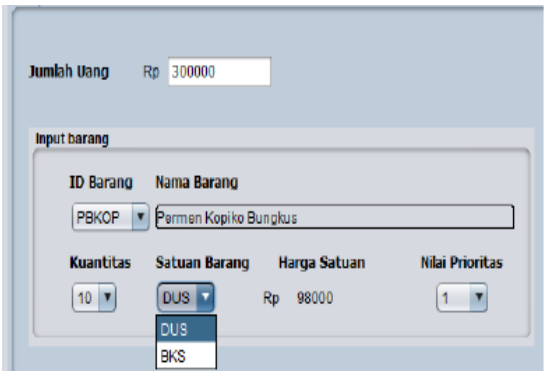


Gambar 2. Form Utama

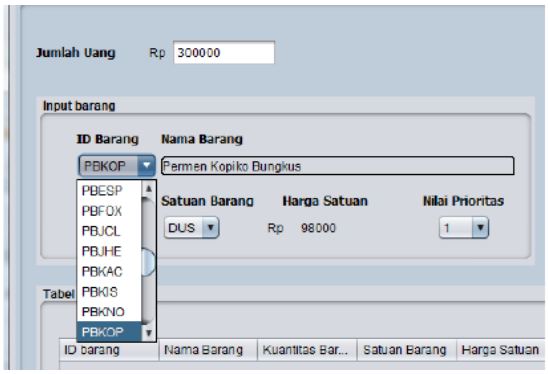


Gambar 3. Form Barang

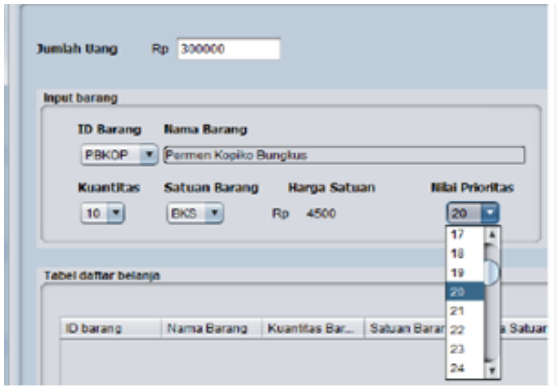
Gambar 4. Form Masalah Baru



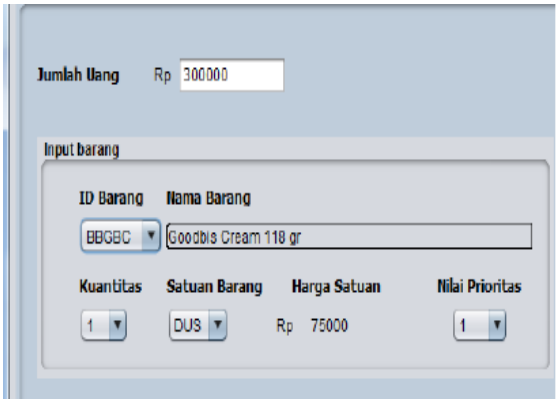
Gambar 8. Form Pilih Satuan Barang



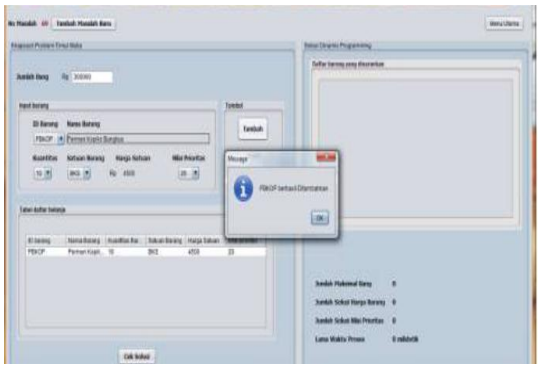
Gambar 5. Form Input Barang



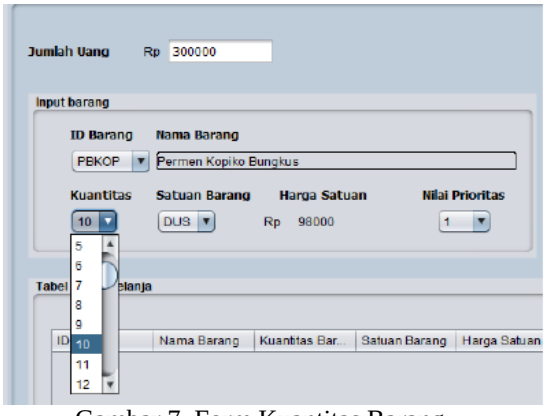
Gambar 9. Form Nilai Prioritas Barang



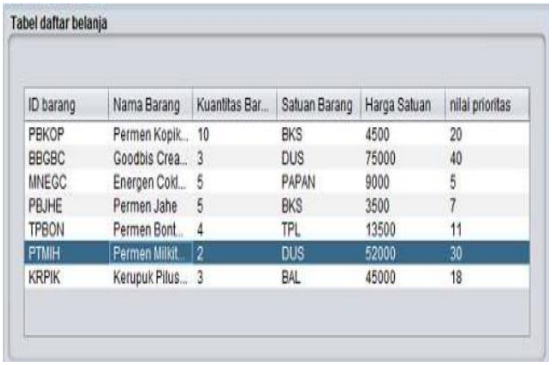
Gambar 6. Form Edit Atribut Barang



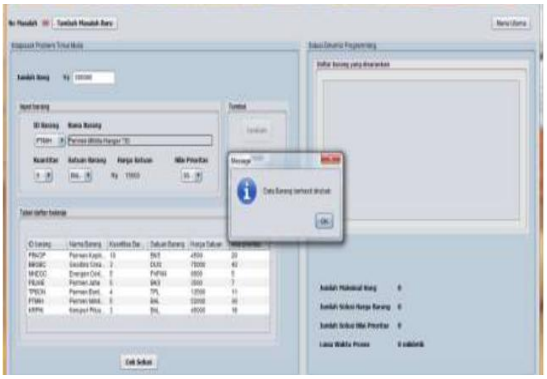
Gambar 10. Penambahan Data Barang



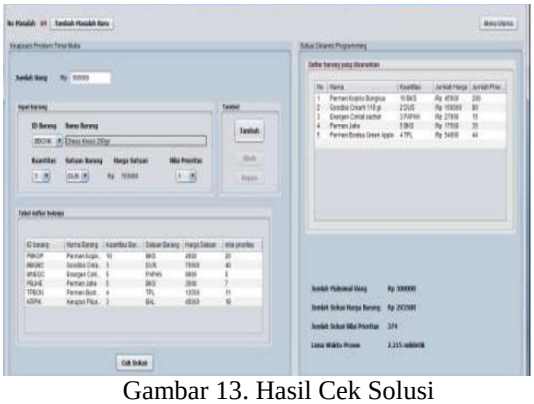
Gambar 7. Form Kuantitas Barang



Gambar 11. Update Data Barang



Gambar 12. Update pesanan Barang



Gambar 13. Hasil Cek Solusi

KESIMPULAN

Hasil penelitian memperlihatkan bahwa penerapan *dynamic programming* untuk menyelesaikan masalah *knapsack problem* khususnya jenis *bounded knapsack problem* dapat memberikan nilai yang positif. Melalui *dynamic programming*, aplikasi yang dihasilkan dapat memberikan nilai optimasi dari daftar barang yang dapat dibeli dengan ketersediaan anggaran belanja yang ada. Aplikasi ini menyediakan fasilitas untuk menyimpan data barang dan data daftar belanja sementara dari pembeli. Pengguna dapat memasukkan *knapsack problem* baru dan mengecek keputusan optimal berupa daftar barang yang disarankan sesuai solusi kombinasi barang yang memiliki jumlah nilai prioritas tertinggi dengan sejumlah batasan dari barang yang dijual.

.

DAFTAR PUSTAKA

[1] Bennett, McRobb, dan Farmer. Object Oriented System Analysis And Design Using UML, Edisi Kedua, McGraw Hill, Berkshire, 2006.

[2] Bhowmik, Biswajit. Dynamic Programming – Its Principles, Applications, Strengths, and Limitations, *International Journal of Engineering Science and Technology*, Volume 2 (9), 4822-4826, India, 2010.

[3] Hasibuan, Zainal A. Metodologi Penelitian Pada Bidang Ilmu Komputer Dan Teknologi Informasi, Konsep: Metode Teknik dan Aplikasi, Depok, 2007.

[4] Kellerer, Hans, Ulrich Pferschy, dan David Pisinger. Knapsack Problems. Springer, Verlag Berlin Heidelberg New York, 2004.

[5] Richard Bellman, " Dynamic Programming ", Princeton University Press.

[6] Rush D Robinett III, " Applied Dynamic Programming for Optimization of Dynamical Systems ", Siam

[7] Sommerville, Ian. Software Engineering - Ninth Edition, Pearson Education Inc., Massachusetts., 2011.

.